Session Code: TLS401

tools & languages

Whidbey

Visual C++ Whidbey Under the Covers: Targeting the CLR

Scott Currie
Program Manager
Microsoft Visual C++
srcurrie@microsoft.com

PDC03

Make the connection

Agenda

- The Basics
- Compiling C++ to MSIL
- Machine Code
- Compilation Modes
- Optimization
- MSIL Linking

CLR Technologies

- MSIL Code
  - MSIL: a portable, stack based representation
  - Easy to verify type safety and memory safety
  - **All MSIL code is compiled by the JIT**
- CLR Data
  - Resides on the Garbage Collecting (GC) Heap
  - Prevents memory leaks and other common problems
  - Metadata encodes all type information

MSIL And C++

- Verifiability is great, but I use pointers!
  - ...in a decidedly unverifiable way!
- MSIL supports:
  - Verifiable data access through metadata
  - First class unverifiable instructions
    - Pointer arithmetic
    - Statically laid out types
    - Access to native heap

Native Heap Access

demo

PDC03

Make the connection

Agenda

- Basics
- Compiling C++ to MSIL
- Machine Code
- Compilation Modes
- Optimization
- MSIL Linking

CLR Types

- Type represented in metadata
- Methods implemented in MSIL
  - Members of the type
  - User specifies scope/visibility
  - Access enforced by runtime
  - Optimizations limited
- Data on managed heap
  - Layout usually determined by runtime
  - Named offsets → No pointer arithmetic

Managed Types

demo

PDC03

Make the connection

Native Types

- Type does not exist in metadata
- Methods or thunks implemented in MSIL
  - Members of the module
  - Globally scoped
  - Access enforced by compiler
  - Fully public → Optimizable
- Data on native heap
  - Layout similar to layout in native codegen
  - Pointer arithmetic → No Verifiability

Native Types Limitations

- #using cannot import native types
  - Data members not exposed
  - Must #include from object's header
- Native types not visible to other languages
- Native member methods awkward to access through reflection

Native Types

demo

PDC03

Make the connection

Agenda

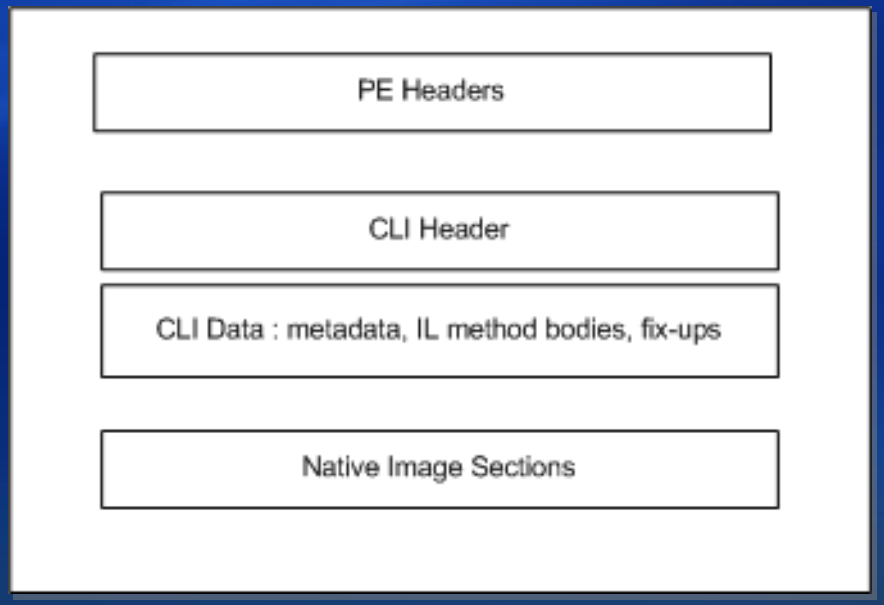
- Basics
- Compiling C++ to MSIL
- Machine Code
- Compilation Modes
- Optimization
- MSIL Linking

Machine Code

- I still want Machine code
  - Performance sensitive or hand tuned code
  - Inline assembly, CPU specific intrinsics
  - Exports to native apps
- Solution: **Mixed Images**

Image Layout

- .NET Assemblies are just PE files
- Additional Data embedded
  - CLR Header
  - Metadata
  - MSIL



- There is still a place for all of the native constructs
- OS Loader

Machine Code

demo

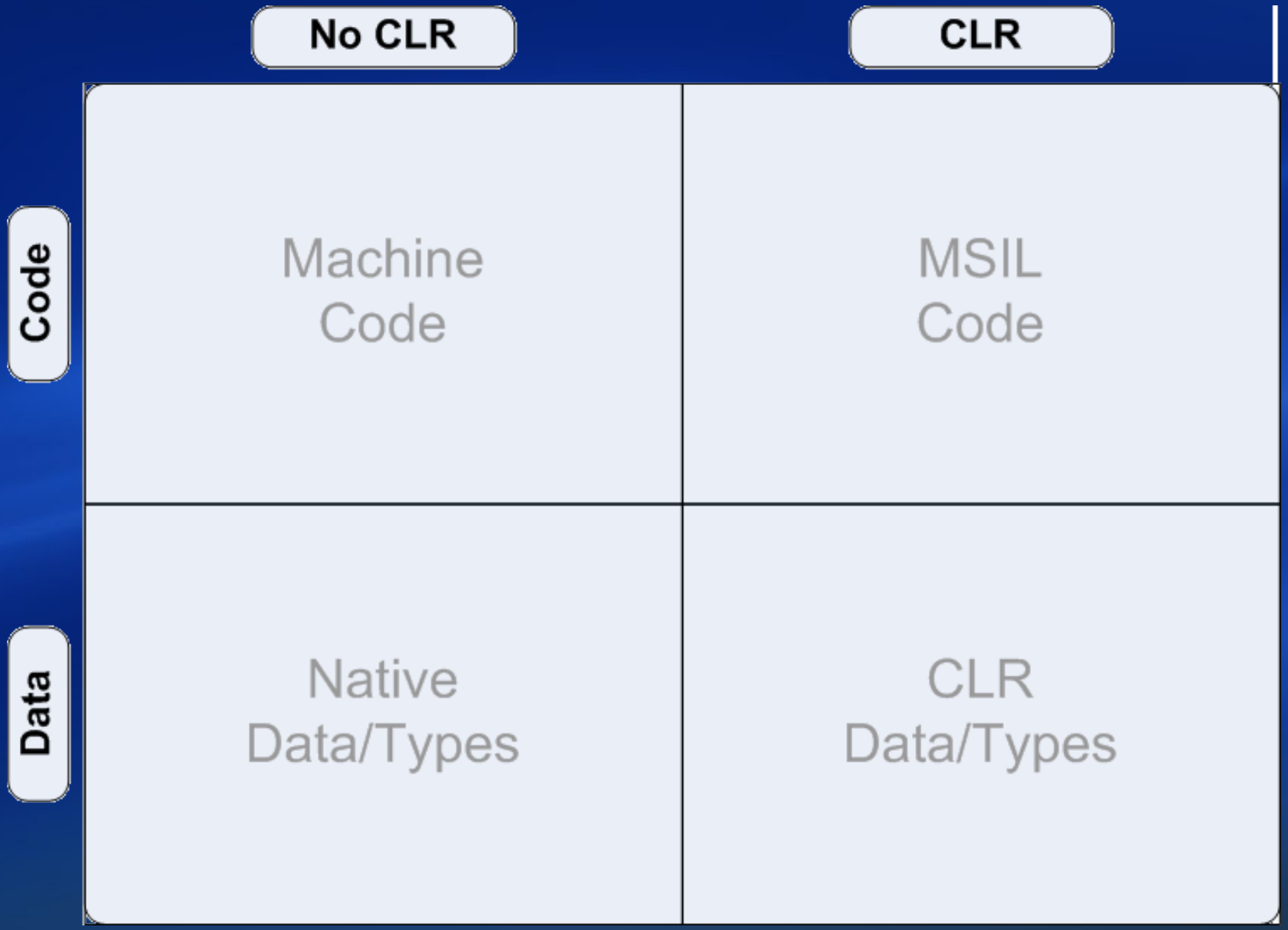
PDC03

Make the connection

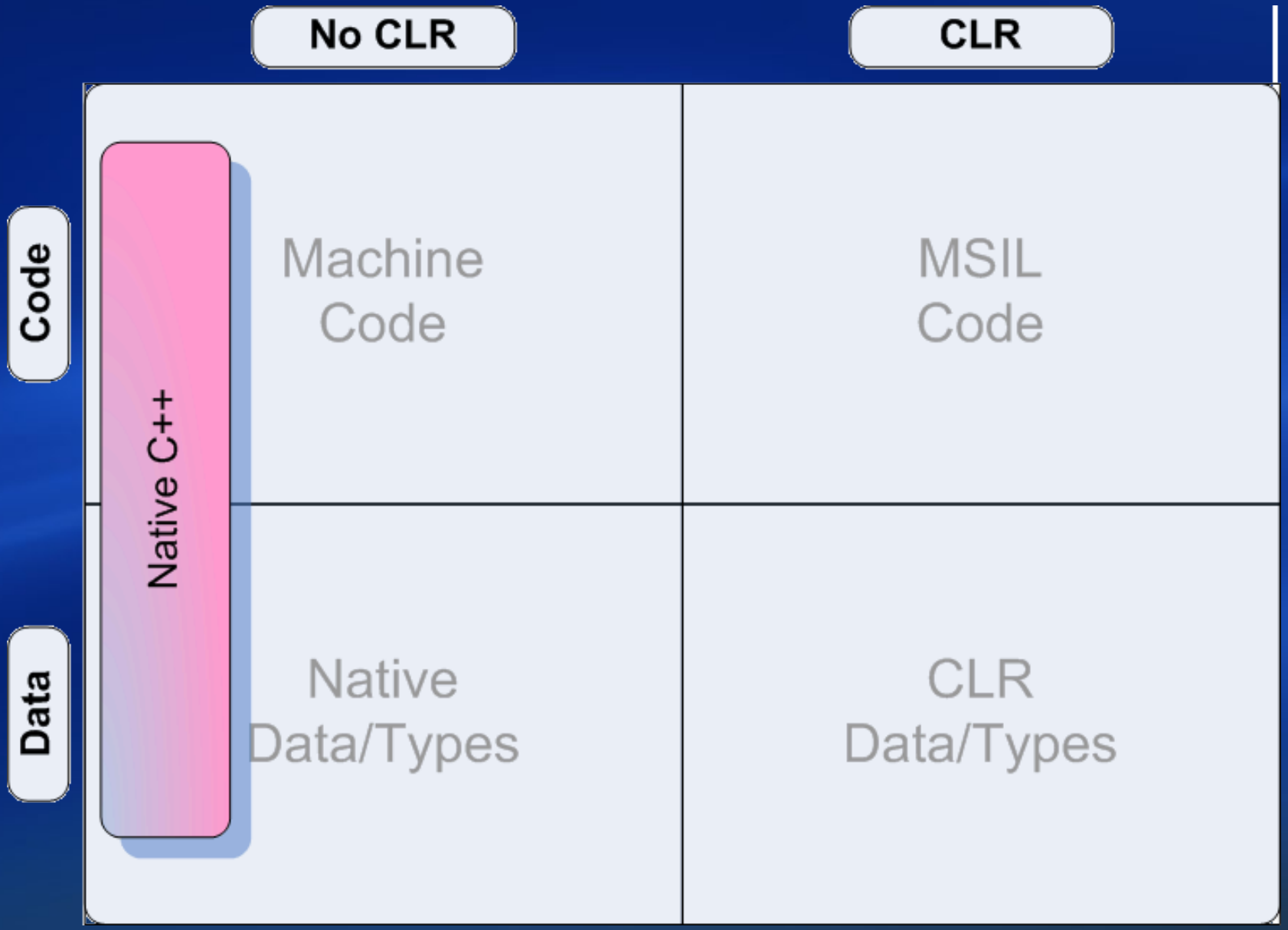
Agenda

- Basics
- Compiling C++ to MSIL
- Machine Code
- Compilation Modes
- Optimization
- MSIL Linking

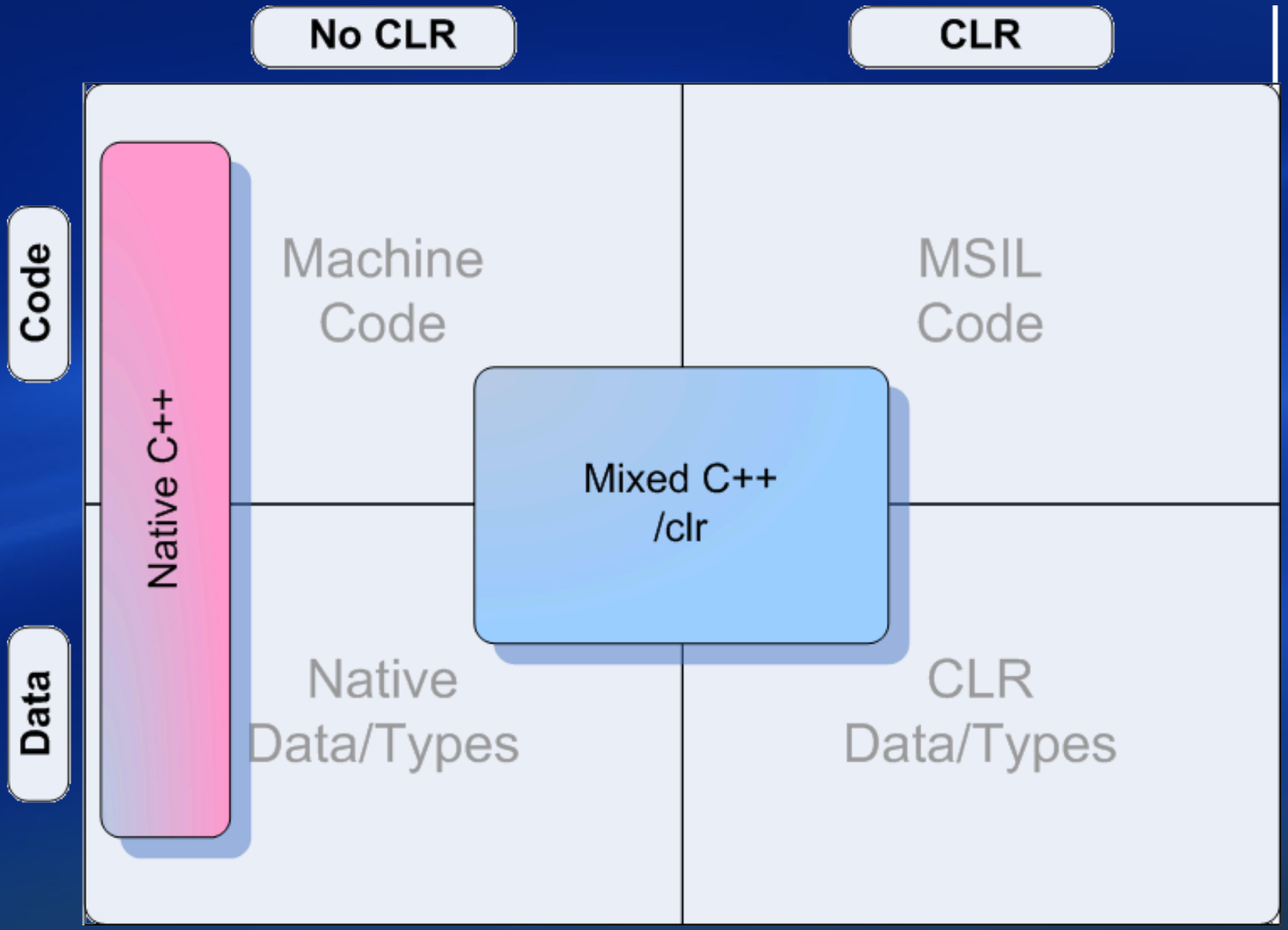
Compilation Modes



Compilation Modes



Compilation Modes



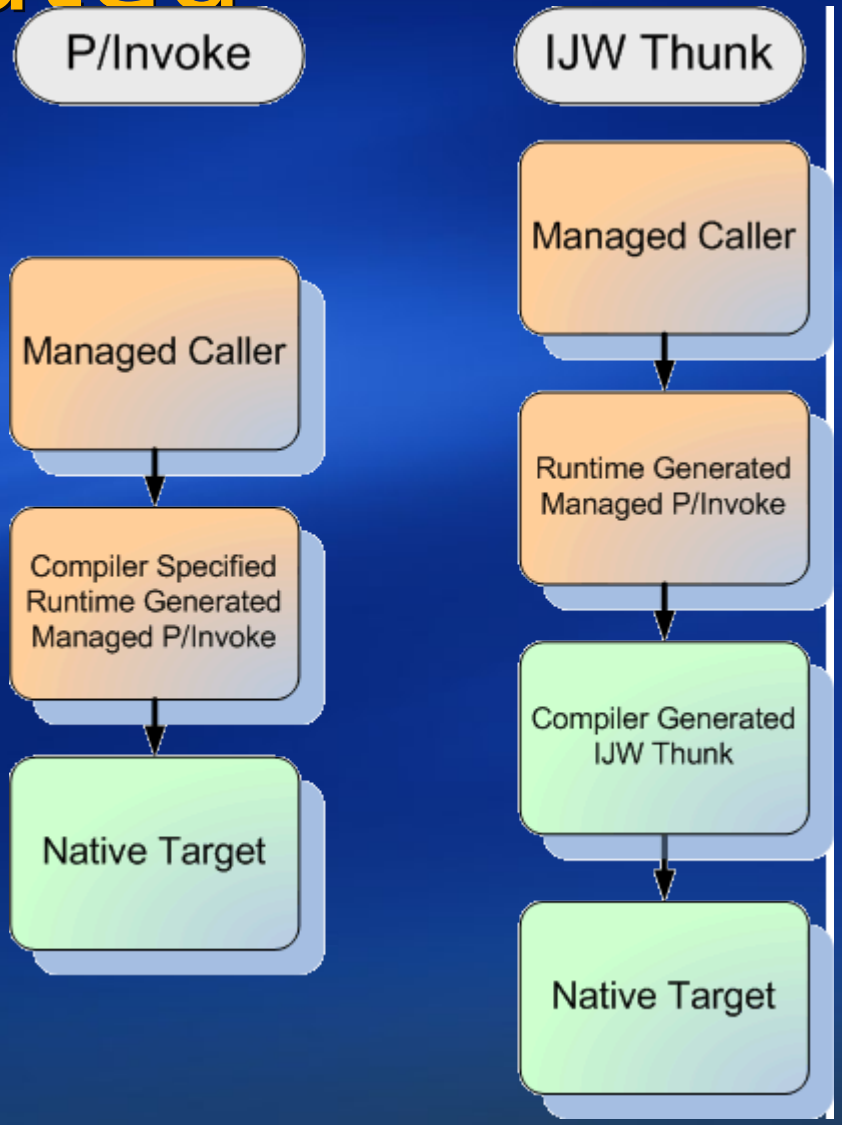
Interop Fundamentals

- All global/native member methods have both unmanaged and managed entry points
  - Actual method implementation
  - Forwarding/Transition thunk
- Method Implementation is MSIL unless:
  - Compiler can’t generate MSIL
    - Inline ASM
    - CPU specific intrinsics
  - User specifies Machine code

.NET↔Native Code Transitions

- .NET code and native code cannot call each other without transitions
  - Calling Convention
  - Type Marshalling
  - GC
- IJW Thunks
  - Compiler provides implementation
  - Call into method through offset or IAT jump
- Managed P/Invokes
  - No method implementation
  - Managed P/Invoke with complete metadata
- Cost

Transition Thunks Illustrated



IJW Thunk Versus P/Invoke

- Intra-assembly
  - IJW Thunks always used
- Inter-assembly
  - IJW Thunk implicitly specified
    - Use #include
  - P/Invoke Thunks explicitly specified
    - Use DLLImport attribute

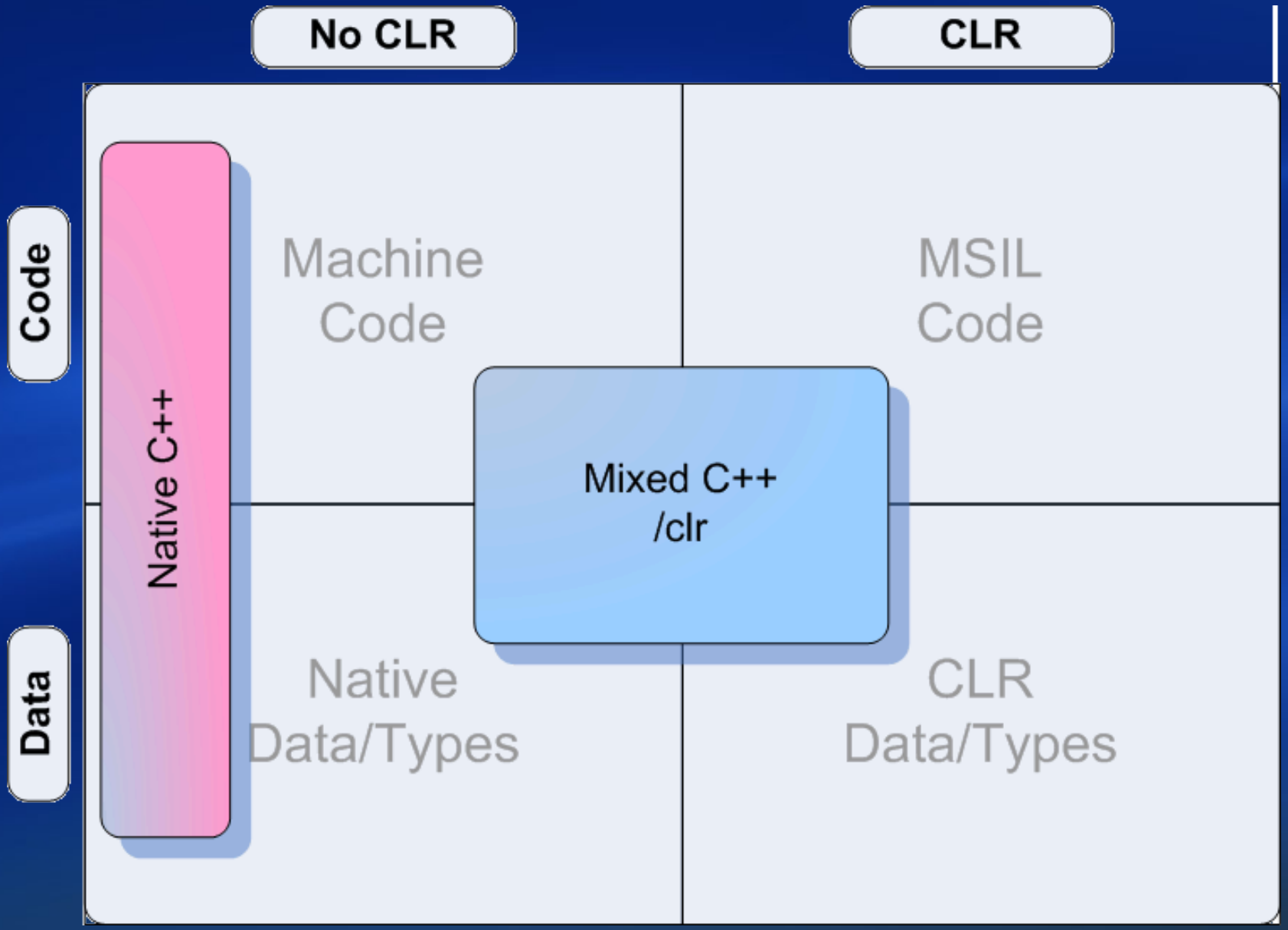
Mixed Code

demo

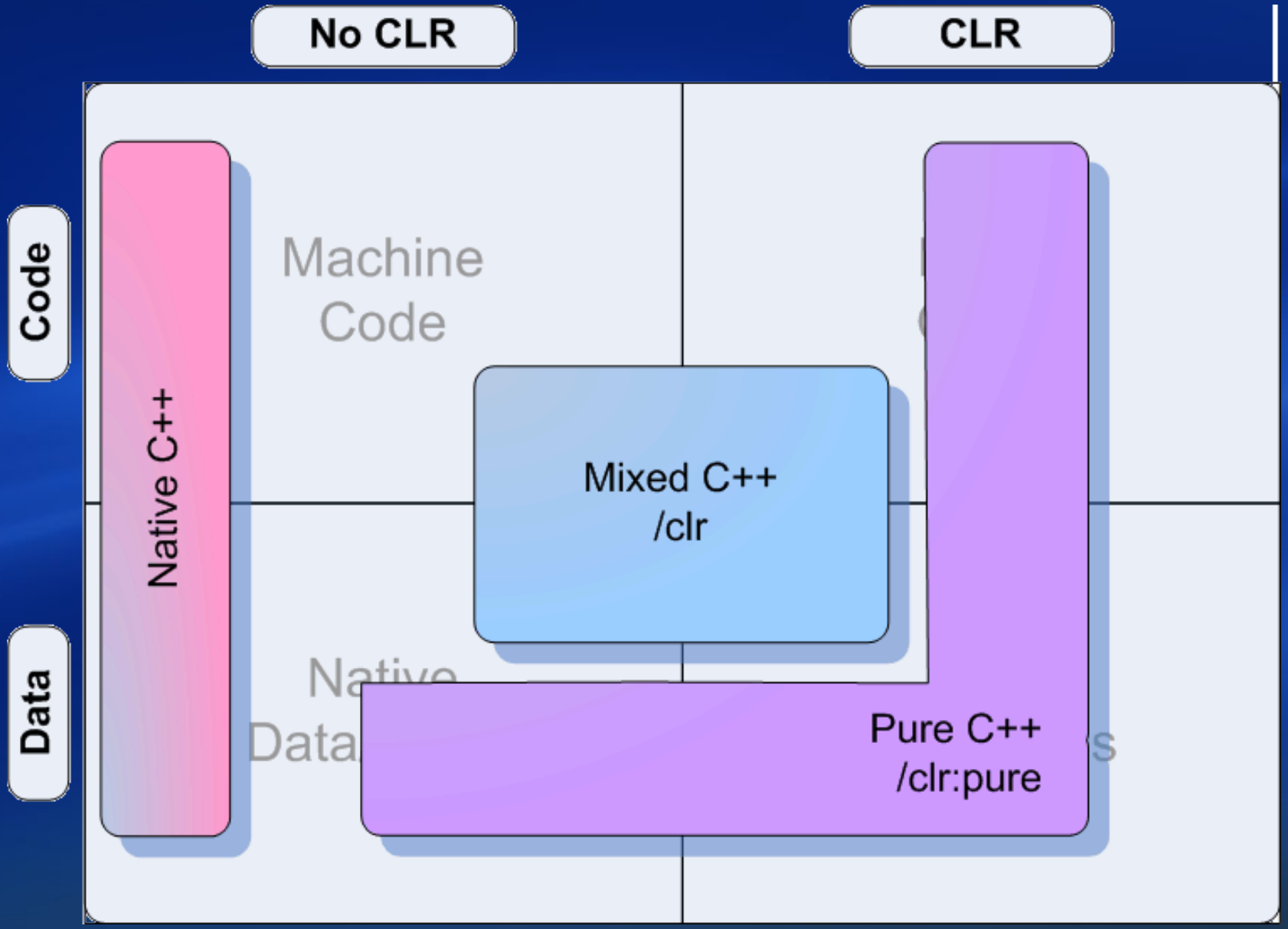
PDC03

Make the connection

Compilation Modes



Compilation Modes



Issues With Mixed Images

- Native image constructs can create problems
  - CLI compliance
  - Native entry point
  - Transition Penalties
  - Static Data
- Solution: **Pure MSIL**
  - Cut out everything native

Pure MSIL

- CLI compliant
- Enables compelling scenarios
  - ClickOnce
  - Form Designers
  - Anything that uses reflection
- Potential performance wins
- Pure code is not necessarily verifiable
- What about interop?

Implicit DLLImport

- Linker fixes up metadata on pure images so that a managed P/Invoke thunk is generated
- Example

```
[DLLImport(msvcrt.dll)]
int printf(const char *format,
...);

int main(void) {
   printf(“Hello World!”);
}
```

```
#include <stdio.h>

int main(void) {
   printf(“Hello World!”);
}
```

Portability?

- Pure Images are CLI compliant
- **BUT,** they are not portable in general
- CRT Initialization
- Native Types
  - Data is statically laid out
  - Different pointer sizes

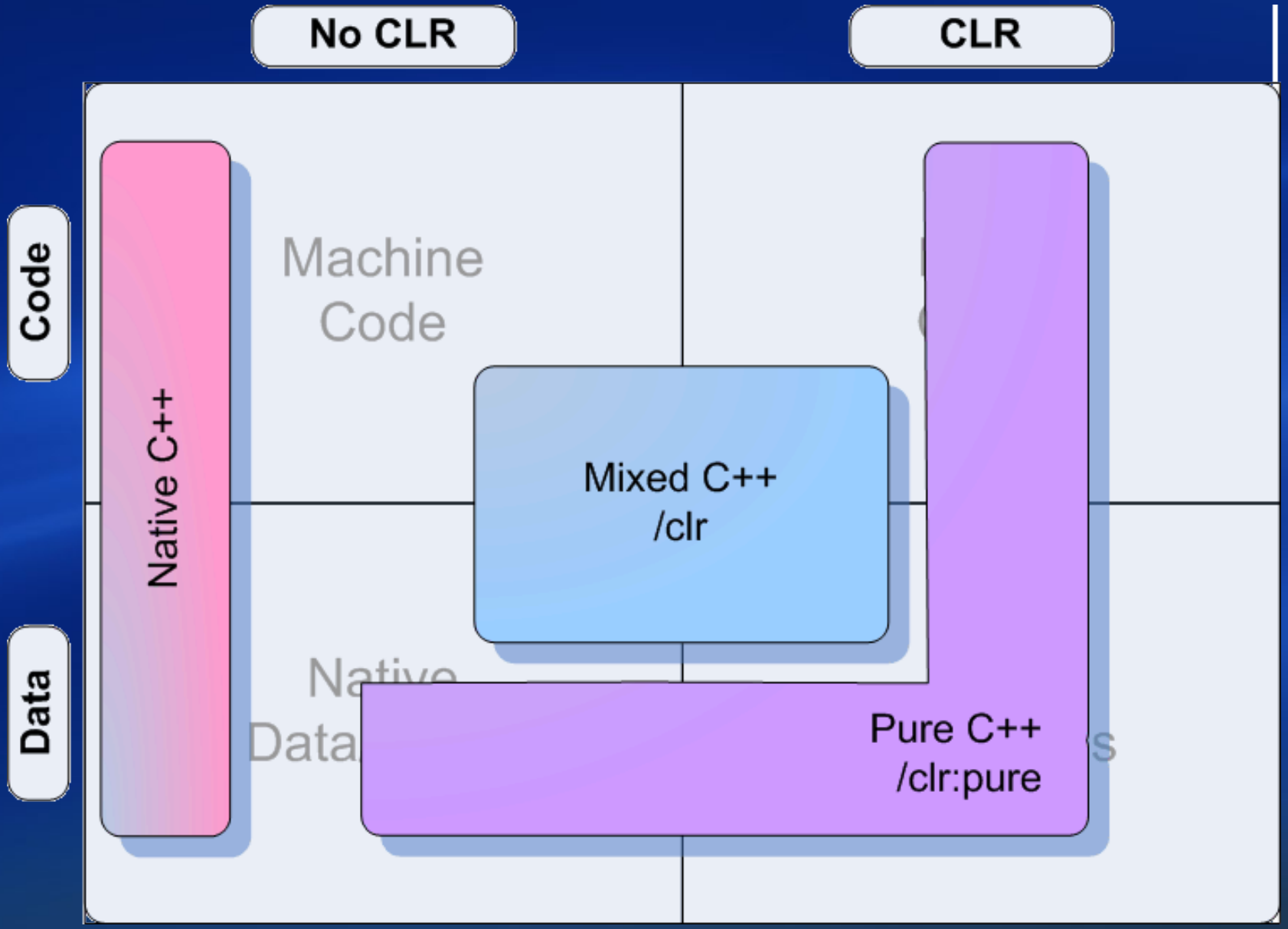
Pure Code

demo

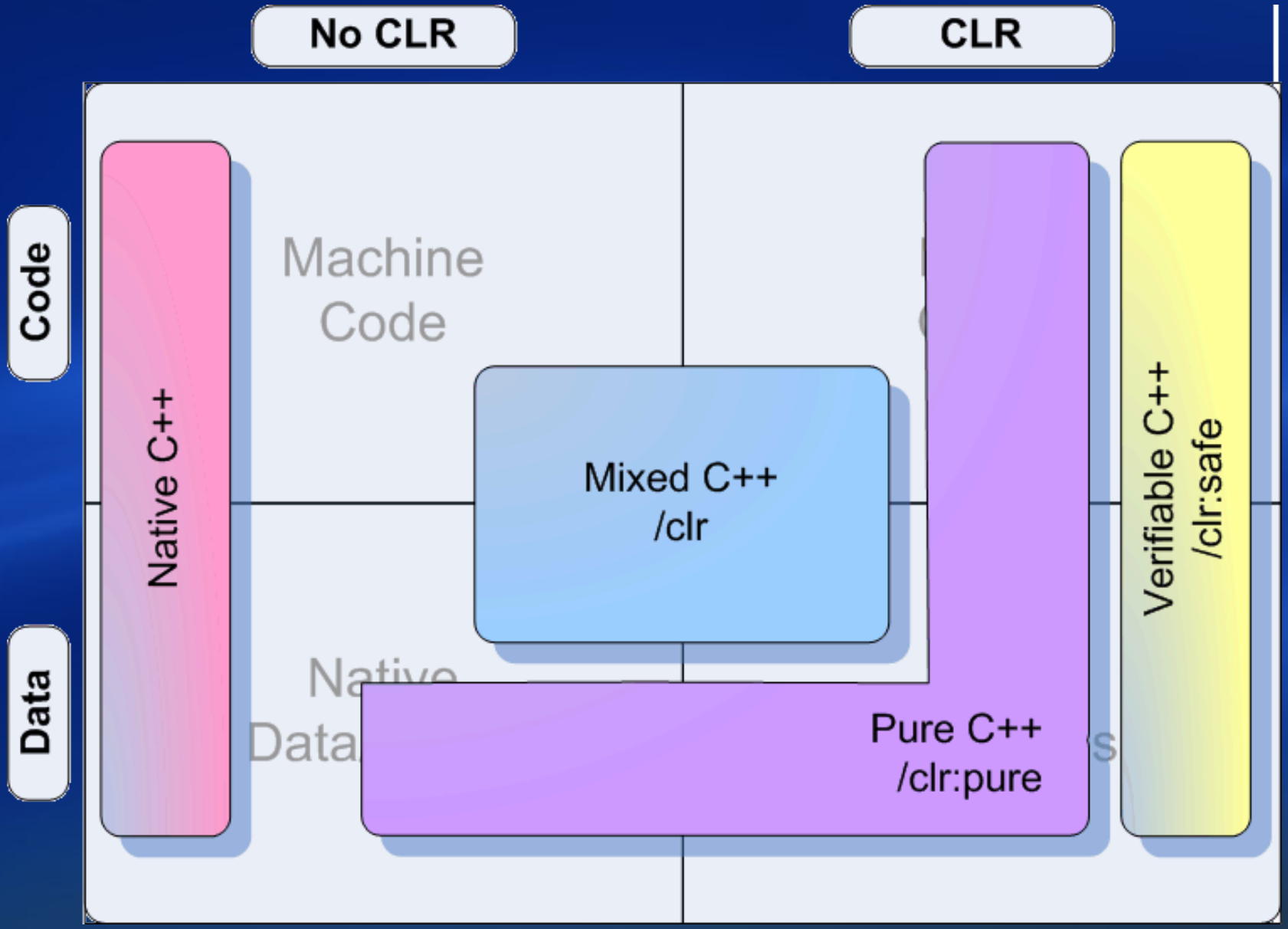
PDC03

Make the connection

Compilation Modes



Compilation Modes



Safe MSIL

- Enables low trust scenarios
  - Remote Code (Web, File Shares, etc.)
  - Yukon
- Ensures that native types are not used
- Potentially more portable
- How it works
  - Forbid all non-verifiable constructs
  - Compiler enforced

Safe Code

demo

PDC03

Make the connection

Agenda

- Basics
- Compiling C++ to MSIL
- Machine Code
- Compilation Modes
- Optimization
- MSIL Linking

Optimization

- Native Code
  - Improvements over 2003
  - “Advanced Code Generation” - 4:45PM
- MSIL for Native Types
  - Improvements over 2003
  - Most opts from native but with different tuning
- MSIL for Managed Types
  - Optimizing for the first time in Whidbey
  - Much happening statically
    - Inlining
    - Expression optimizations
    - Loop optimizations

Verifiable Inlining

```
#using <mscorlib.dll>

__gc class Type {
private: int hidden;
public:  int Accessor() { return this->hidden; }
};

int main() {
      Type *t = new Type();
      return t->Accessor();
}
```

```
/********************* Inlining **********
 IL_0004:  newobj    instance void Type::.ctc
 IL_0005:  stloc.0
 IL_0006:  ldloc.0
 IL_0007:  ldfld     int32 Type::hidden
 IL_0008:  ret
```

Verifiable Inlining: Solutions

- Analyze for legal inlining opportunities
  - Visibility
  - Security Boundaries
  - Attributes
- Relax above restrictions where possible

MSIL Strength Reduction

```
#define MAX_ARRAY 50

int main() {
      int a[MAX_ARRAY];
      for (int i=0; i<MAX_ARRAY
            a[i] = 0;
      }
      return 0;
}
```

```
 IL_0002:  ldloc.0
 IL_0003:  ldc.i4.4
 IL_0004:  mul
 IL_0005:  ldloca.s  V_1
 IL_0007:  add                 //
a[i]
 IL_0008:  ldc.i4.0
 IL_0009:  stind.i4            //
a[i] = 0;
 IL_000a:  ldloc.0
 IL_000b:  ldc.i4.1
 IL_000c:  add                 // i+
```

```
#define MAX_ARRAY 50
int main() {
      int *a = new int[MAX_ARRA
      for (int *end = a+MAX_ARR
{
            a = 0;
      }
      return 0;
}
```

```
 IL_0003:  ldc.i4.4
 IL_0005:  ldloca.s  V_1
 IL_0007:  add
 IL_0008:  stloc.1             // a+
+;
 IL_0009:  ldloc.1
 IL_0008:  ldc.i4.0
 IL_0009:  stind.i4            // a =
0;
```

Agenda

- Basics
- Compiling C++ to MSIL
- Machine Code
- Compilation Modes
- Optimization
- MSIL Linking

MSIL Linking

- Link.exe can now consume .netmodules
- Permits multiple languages in same single-file assembly
- Dependencies can be created among modules and objects written in different languages
- No more MSIL roundtripping

Linker Consumable Files

- Object Files
  - Native C++
  - Mixed C++ (/clr)
  - Pure C++ (/clr:pure)
  - Safe C+ (/clr:safe)
- .netmodules
  - C#, VB, etc.
  - Pure C++ (/clr:pure)
  - Safe C++ (/clr:safe)

MSIL Linking

demo

PDC03

Make the connection

Summary

- Mixed code enables gradual transition to WinFX
  - Be mindful of transition penalties
- Pure code enables variety of tools based scenarios
- Safe code enables low trust scenarios
- Optimization of MSIL improves performance
- MSIL Linking improves multi-language assemblies

Additional Resources

- Related sessions
  - Leveraging Native Code with .NET (TLS311) – Wed - 11:30AM / 150,151
  - Advanced Code Generation (TLS400) – Today - 4:45PM / Room 408AB
- Send us feedback!
  - srcurrie@microsoft.com
  - vswish@microsoft.com
  - Public newsgroups
    - microsoft.public.vc.*
    - microsoft.public.dotnet.languages.vc.*
- Other resources
  - VC++ Hands On Lab Here at the PDC
  - Ask The Experts
  - VC++ Product Booth

Microsoft®

Make the connection

Microsoft Professional Developers Conference 2003

October 26 - 30, 2003, Los Angeles, CA

Microsoft